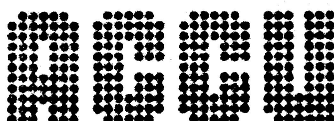
DATUM:

19 maart 1976



FLODDER 20

AUTEUR:

H. Koppelaar

TITEL: Inleiding in Lisp.

Voorwoord

De voorliggende inleiding is geschreven voor de Utrechtse gebruiker van de Cyber. In Utrecht is aanwezig LISP 1.5 zoals oorspronkelijk gedefinieerd door McCarthy. De Utrechtse gebruiker zij gewaarschuwd voor afwijkingen met de Delftse LISP van Prof. v.d. Poel en de Amsterdamse LISP van Champeaux. Deze inleiding is zeer praktijkgericht en behandeld dus niet de meta-taal voor LISP.
Verder is deze inleiding, in afwijking van alle inleidingen die ik onder ogen kreeg, gericht op lijstverwerking (list-processing).
Immers de gebruiker is niet gediend met voorbeelden voor numerieke zaken waartoe deze taal niet geconcipieerd is.
Met dank aan Jan van Kats voor de inplementatie van LISP.

1. Inleiding.

De programmeertaal LISP heeft als voornaamste eigenschap dat zij werkt met "lijsten" (=lists), vandaar de naam "lijst-verwerker" (=listprocessor, kortweg: LISP). Een list wordt geschreven als een "rij-uitdrukking" (=string-expression, kortweg: S-expression), bijvoorbeeld de letters A, B en C in een list

(A B C)

Op de plaatsen tussen de letters hadden ook komma's mogen staan. De haakjes in S-expressions zijn altijd gepaard, bijvoorbeeld ((A B C) is geen S-expression, maar ((A B) C) wèl.
De programma's in LISP worden zelf ook geschreven in S-expressions.
Programma's in LISP kunnen bestaan uit primitieve functies werkend op lists, maar ook uit samengestelde functies werkend op lists.

2. Primitieve functies.

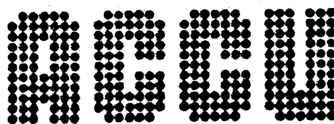
We schrijven een horizontale pijl waarvan rechts het resultaat komt van de werking van een primitieve functie. Iedere functie heeft één of meer argumenten nodig, deze schrijven we in een S-expression rechts van de functienaam, bijvoorbeeld de functie CAR die het eerste element uit een list pakt

((CAR A) →A, immers de lijst bevat slechts het 1e element
(CAR (A B))→A, immers A is de eerste van (A,B)
(CAR (KIE RE) WIET)→ (KIE RE)

Er zit ook een functie CDR in het LISP-systeem die alles behalve de eerste van een list pakt

(CDR (A B))→ B
(CDR (KIE RE) WIET)→WIET

DATUM: 19 maart 1976



FLODDER 20

AUTEUR: H. Koppelaar

TITEL: Inleiding in Lisp.

De functie CONS bindt twee losse elementen tot een list

```
(CONS AAP NOOT)→ (AAP NOOT)
(CONS X (Y RE))→ (X (Y RE))
(CONS X Y Z     → wrong nr. of arguments
```

De functie EQ vergelijkt twee losse elementen en geeft "true" (=T) als deze twee gelijk zijn

```
(EQ X X)→*T*,  omdat de naam T de waarde *T* heeft
(EQ X Y)→NIL,  omdat de naam F (van "false") de waarde NIL heeft
```

Tot nu toe spraken we over "losse elementen", in LISP moeten die "atomen" heten. Er is een primitieve functie ATOM die test of een element een atoom is

```
(ATOM AAP)→*T*
(ATOM (AAP NOOT))→ NIL,want argument is list
(ATOM (AAP))→ NIL, want argument is list
```

Er is een faciliteit om diverse CAR en CDR samen te voegen, als volgt

```
(CADR (A B C))→B
(CADDR (A B C))→C
```

Het is mogelijk om een variabele de waarde van een lijst te geven, dit kan met de functie CSET, bijvoorbeeld : we geven de "onbekende" variabele X de waarde (A,B,C), als volgt

```
(CSET X (A,B,C))→X
```

Het LISP-systeem geeft dan, als antwoord ten teken dat deze naamgevingsoperatie volvoerd is de naam van de variabele, in dit geval X, terug. We kunnen nu testen of X een lege list vertegenwoordigt, dit gaat met de functie NULL.

```
(NULL X)→ NIL, immers X is niet leeg.
```

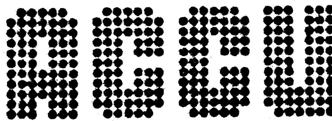
De bekende "als....dan...." uitdrukking heeft in LISP de volgende gedaante

```
(COND ((predikaat)(doe iets))
```

Een predikaat is een statement die true of false oplevert, dus (NULL X) is er zo een. We kunnen met COND (afkorting van "condition") bijvoorbeeld vragen om te testen of de list X leeg is en zo ja, dan moet de functie COND aan X de list (A,B,C) als waarde geven. Dit gaat als volgt

```
(COND  ((NULL X)(CSET X'(A,B,C))))
```

In gewoon nederlands staat hierboven: als (NULL X) waar is doe dan (CSET X (A B C))



AUTEUR:

H. Koppelaar

TITEL: Inleiding in Lisp.

COND kan een willekeurig aantal argumenten hebben, bijvoorbeeld

```
(COND  ((ATOM X) X)
       (T  (CAR X)))
```

Hierboven staat in gewoon nederlands: als X een atoom is, geef dan X op, zoniet geef dan als true waar is, de eerste van X op. Het zal duidelijk zijn dat true altijd waar is, dus als X geen atoom is, wordt er altijd de eerste van de list, waarvoor X staat, opgeleverd.

(

3. De programmaopbouw voor LISP.

Voordat we een LISP-programma in executie kunnen brengen op de computer, moeten we nog iets weten over de vertaler voor LISP. Uiteraard kan de computer niet zonder meer "begrijpen" wat wij aan programma's aanbieden maar vertaalt hij het eerst met geëigende vertalers. De LISP vertaler noemt zichzelf EVALQUOTE en deze heeft altijd twee argumenten nodig. Dus als we (CAR A), zo ook met (CAR (A,B)), dit moet voor de aanbieding worden CAR((A,B)).
Dit verschuiven van het eerste haakje voorbij de naam van de eerste aan te bieden functie, heeft tot gevolg dat EVALQUOTE twee argumenten ziet, n.l. de naam van een functie, in dit geval CAR en de argumentenlijst van die functie. Terwijl (CAR A) er voor EVALQUOTE uitziet als één argument, vanwege het list-haakje.
Als we op deze manier consequent bij elke eerste functie die we EVALQUOTE aanbieden het linkerhaakje verschuiven, werkt LISP. Een voorbeeldprogramma luidt:

```
KOPHE,T4,CM45000.
ACCOUNT,xxxxxxxxx.
ATTACH,LISP,ID=UACCU.
RFL,45000.7 LISP,L.
7/8/9
______blanco kaart______

    CAR(A)   CAR((A,B))
    CDR((A,B))  CAR(((KIE RE)  WIET))
STOP ))))))
FIN
6/7/8/9
```

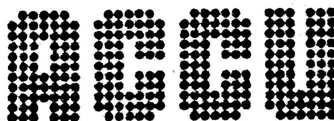
Let op de stop en de fin-kaart. Op de eerste kaart van dit karwei, die hier blanco genoemd is, mag commentaar staan. In de ponskaart voorbij kolom 72 mag geen programmatekst staan, de kolommen 73 t/m 80 kunnen gebruikt worden voor kaart-nummering.

4. Het samenstellen van functies.

Met louter een pakket van primitieve functies, zoals besproken in sectie 2, is in vele gevallen een gebruiker onvoldoende toegerust voor de toepassing van LISP.

DATUM:

19 maart 1976



FLODDER 20

AUTEUR:

H. Koppelaar

TITEL: Inleiding in Lisp.

Om met behulp van in LISP gedefiniëerde primitieve functies meer ingewikkelde zaken te kunnen doen, bestaat er een functie DEFINE waarmee de benodigde extra in te bouwen functies gemaakt (gedefiniëerd) kunnen worden. Deze functie kan meerdere functies tegelijk definiëren, dat gaat als volgt:

```
(DEFINE (lijst van functies))
0       1                 10
```

De lijst van functies op zijn beurt kan één of meerdere elementen bevatten:

```
((functie1)(functie2).....(functien))
12        22        2     2        21
```

Iedere functie op zichzelf moet er zo uitzien:

```
(naam (wat doet 'ie))
2     3            32
```

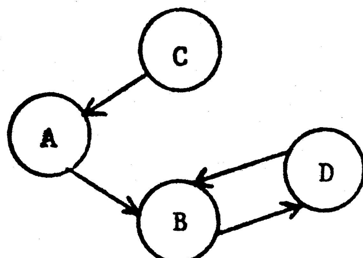
De taak van een functie is wat 'ie doet, deze heeft altijd de volgende vorm:

```
(substitueer (argumentenlijst)(body))
3            4                44    43
```

De betekenis van de opdracht "substitueer" is het contact met de "buitenwereld" voor de te definiëren functie. Immers de opdracht "substitueer" kan ervoor zorgen dat te bewerken elementen, mits ze in de argumentenlijst gegeven worden, in de body gesubstitueerd worden, teneinde daar een bedoelde bewerking te ondergaan. Bijvoorbeeld: we maken een functie met de naam TA, die van het element A het element B, van B een D, van C een A en D een B maakt als volgt:

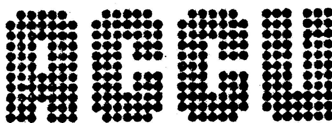
```
      A B C D
TA:  ↓
      B D A B
```

Een diagram van de mogelijke overgangen kan er zo uitzien:



DATUM:

19 maart 1976



FLODDER 20

AUTEUR:

H. Koppelaar

TITEL: Inleiding in LISP.

Willen we nu deze functie TA maken, dan zal de body van deze functie zodanig moeten zijn, dat als een A, B, C of D als argument ingestopt wordt (d.m.v. de functie "substitueer"), dat dan de geëigende letter resp. B, D, A of B eruit komt. In de body stoppen we:

```
(COND
4
   (( EQ ELEMENT (QUOTE A))(QUOTE B))
   56            7      766       65
   (( EQ ELEMENT (QUOTE B))(QUOTE D))

   (( EQ ELEMENT (QUOTE C))(QUOTE A))

   (( EQ ELEMENT (QUOTE D))(QUOTE B))
```

Dus we testen met de primitieve functie of ELEMENT gelijk is aan A letterlijk en zoja, dan geeft COND letterlijk B terug, en zo voor B, C of D. Stel nu dat de voor ELEMENT gegeven waarde (dat is het te transformeren symbool) geen A, B, C of D is, dan transformeren we niet en laten we melden: ONMOGELIJK.
Daartoe voegen we een laatste regel aan de body toe die zegt: "als TRUE waar is geef dan het atoom ONMOGELIJK", als volgt:

```
(T ONMOGELIJK))
5            54
```

Let op de sluithaak op niveau 4 voor de COND functie. Rest nog te vermelden dat de LISP term voor "substitueer" in het hier behandelde verband LAMBDA is, dan wordt de volledige functie TA in LISP geschreven als

```
(DEFINE ((TA (LAMBDA (ELEMENT)(COND

   ((EQ ELEMENT (QUOTE A))(QUOTE B))

   ((EQ ELEMENT (QUOTE B))(QUOTE D))

   ((EQ ELEMENT (QUOTE C))(QUOTE A))

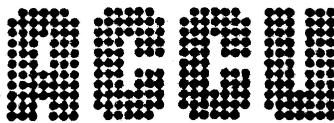
   ((EQ ELEMENT (QUOTE D))(QUOTE B))

   (T ONMOGELIJK))) )))
```

Let weer op de sluithaken en probeer te begrijpen wat er gebeurt als het te transformeren element noch een A, B of C noch een D is, (de computer loopt de body door in de volgorde waarin deze gegeven is).
Gebruik nu de ponskaarten van sectie 3 en voeg de hierboven uitgeschreven functie-definitie toe, let op 't verschuiven van de eerste haak voor de naam DEFINE en roep vervolgens deze functie aan door een kaart tussen te voegen met:
(TA(A),TA(C),TA(E).
Deze kaart komt nu natuurlijk nà de functie-definitiekaarten.

DATUM:

19 maart 1976



FLODDER 20

AUTEUR:

H. Koppelaar

TITEL: Inleiding in Lisp.

5. Recursieve definitie van functies.

Het definiëren van functies op de wijze zoals uitgelegd in sectie 4 heeft het nadeel dat voor sommige toepassingen de definitie-zelf erg omvangrijk wordt. Neem bijvoorbeeld de functie TA voor 26 argumenten:

```
         | A B C D E F G H I ......Z
    TA:  ↓
           B D A B Z Y X W V ......E
```

Voor deze functie wordt de definitie met behulp van de in sectie 4 behandelde methode meer dan 26 regels lang. Dit is trouwens geen pathalogisch voorbeeld. Om efficiënter te kunnen programmeren introduceren we en ander type definitie van functies, n.l. de recursieve definitie. Bij dit soort definitie roept de functie zichzelf één of meerdere keren aan om een (lange) lijst van inputs te verwerken. Om met een voorbeeldje te beginnen: een functie die het eerste atoom in een lijst vindt, ongeacht de hoeveelheid haken waarmee dat atoom omgeven is, noemen we **FF** en kunnen we als volgt definiëren

```
(DEFINE ((FF (LAMBDA (X)(COND

    ((ATOM X) X)

    (T (FF (CAR X))) ))) )
```

Het kan onmiddellijk duidelijk zijn wat hier staat, als het programma zo gelezen wordt: test eerst of de voor X substitueren lijst soms al een los atoom is, zoja (=true) geef dan gelijk aan de functie de waarde van dit atoom, zonee (=false) ga dan altijd door (=true) met het toepassen van de functie op de eerste van de lijst. De werkwijze van LISP voor een aanroep van FF op een lijst (((DRS)P)) is dus als volgt (((DRS)P)) → ((DRS)P) → (DRS) → DRS.
Hiermee is tevens aangetoond dat (ATOM X) als stopcriterium dient, immers als een recursief gedefiniëerde functie geen stopcriterium heeft dan is de definitie incorrect.
Als volgende voorbeeld nemen we eerst een functie "member", die kan uitzoeken of een op te geven atoom in een te specificeren lijst zit. De bedoeling is dat member *T* geeft als het atoom daadwerkelijk in die lijst zit, dat kunnen we zo definiëren:

```
(MEMBER (LAMBDA (E L)(COND (ZIT ER NIET IN)
2
((NULL L)((EQ E (CAR L))T) ZIT ER WEL IN)

(T (MEMBER E (CDR L))) )))
                          2
```

TITEL: Inleiding in Lisp.

Let in deze functie op 't uitgebreide stopcriterium, immers voor het geval dat het in E te substitueren element daadwerkelijk in de lijst (die in L gesubstitueerd wordt) zit geldt dat de vergelijking van dat element met het betreffende element uit de lijt *T* gaat opleveren, echter indien dat gelijkluidende element in de lijst niet te vinden is, dan zal member die lijst blijven afstropen totdat die lijst leeg is en dan geeft het eerste deel van het stopcriterium ((NULL L) F) een melding.
Ter verdere illustratie geven we een voorbeeld van de functie EQUAL die twee lijsten kan vergelijken, een uitbreiding van EQ dus. Bestudeer deze:

```
(EQUAL (LAMBDA (X Y)(COND
2
  ((ATOM X)(COND((ATOM Y)(EQ X Y))(T F)))
  ((EQAUL (CAR X)(CAR Y))(EQUAL (CDR X)(CDR Y)))
  (T F))) )
          2
```

Laten we nog eens enkele functies de revue passeren. Als eerste een generalisatie van de hiervoor behandelde functie TA. Deze functie TA immers had het bezwaar dat indien daarmee lange lijsten atomen getransformeerd moesten worden, dat dan de functie-definitie evenredig in omvang zou toenemen. We gaan nu generaliseren zodanig, dat willekeurige (te specificeren) lijsten in willekeurige (te specificeren) lijsten van gelijke lengte getransformeerd kunnen worden. Dat gaat zo:

```
(ASHBY (LAMBDA (EL X Y)(COND
2
    ((NULL X)(QUOTE GEEN INPUT))
    ((NULL Y)(QUOTE GEEN OUTPUT))
    ((EQ EL (CAR X))(CAR Y))
    (T (ASHBY EL (CDR X)(CDR Y))) )))
                                  2
```

De functie ASHBY is een generalisatie van de hiervoor behandelde TA, dit kunnen we zien door aan te roepen ASHBY(A (A,B,C,D,E)(B,D,A,B,E)).
Een volgend voorbeeld is de functie REVERSE, deze zou zonder recursieve definitie onmogelijk te maken zijn.

```
(REVERSE (LAMBDA (X Y)(COND

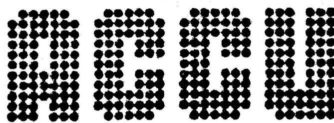
    ((NULL X) Y)
    (T (REVERSE (CDR X)(CONS (CAR X)Y))) )))
```

De executie van een aanroep REVERSE(A (B,C)) levert op REVERSE(NIL (A B C)) →(A,B,C)
De executie van een aanroep REVERSE((A,B)c) verloopt zo:

REVERSE(B (A C))→ REVERSE (NIL(B AC)) → (B,A,C).

DATUM:

19 maart 1976



FLODDER 20

AUTEUR:

H. Koppelaar

TITEL: Inleiding in Lisp.

Tenslotte de executie van REVERSE((A,B,C)D,E), deze levert op

(C,B,A,D,E). Ga dit zelf na!

Tenslotte vermelden we nog een functie die kan substitueren, ga zelf na wat deze precies doet:

```
(SUBST (LAMBDA (EL Y Z)(COND

    ((EQUAL Y Z) EL)

    ((ATOM Z) Z)

    (T (CONS (SUBST EL Y (CAR Z))

             (SUBST EL Y (CDR Z))) ))) )
```

Niet al deze functies behoeven gedefiniëerd te worden om te kunnen testen. Immers LISP zelf bevat heel wat functies, om achter de namen te komen van de functies die erin zitten, roepe men de "objectenlijst" van LISP als volgt aan:

```
EVAL(OBLIST NIL)
```

en dan meldt EVALQUOTE de hele lijst. Voor de betekenis van al deze functies zie [ ].
In de OBLIST (=objectenlijst) zit ook een functie RANDOM. Om het i-de element in een lijst X te vinden, waarbij i=y*lengte(X) en Y random gekozen is, kan men als volgt de functie randomelement definiëren:

```
(RANDOMELEMENT (LAMBDA (X I)(COND

    ((ONEP (LENGTH X))(CAR X))

    ((LESSP (TIMES (LENGTH X)Y) 1.0)(CAR Y))

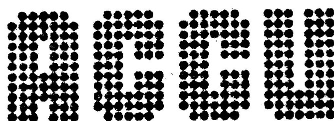
    (T (RANDOMELEMENT (CDR X) Y))) ))
```

De aanroep wordt nu: RANDOMELEMENT((A,B,C)(RANDOM 0)). RANDOM 0 levert een getal in het interval [0,1].

6. Functies werkend op functies.

Om contrôle te krijgen over het herhaald gebruik van functies in een programma, kunnen we functies maken die te controleren functies als input hanteren.

DATUM:
19 maart 1976



FLODDER 20

AUTEUR:
H. Koppelaar

TITEL: Inleiding in Lisp.

Een voorbeeld hiervan is een functie die n-maal de functie TA kan laten werken op een argument, door telkens de output van de vorige bewerking van TA weer als input aan TA te geven voor de volgende bewerking. We noemen deze functie COMPOSE:

```
(COMPOSE (LAMBDA (FN X N)(COND
2
    ((ZEROP N) X)

    ((NULL X) GEEN BEWERKING MOGELIJK)

    (T (COMPOSE FN (FN X)(SUB1 N))) )))
```

Een aanroep is nu

```
COMPOSE ((FUNCTION TA)A 2)
```

Had dit ook gekund met MAPLIST uit de OBLIST? Veel plezier met EVALQUOTE.